NXP CUP
3D Printed Kit Guide

Table of contents

1. Motivation

This project addresses the shortage and high price of car kits by offering an affordable solution. Through 3D printing, we aim to make these kits more accessible for students. This is preferable because many universities now provide students with access to 3D printers. By enabling teams to 3D print their own kit, we ensure that everyone has the resources they need to compete.

Additionally, past NXP Cup competitions have shown that custom 3D printed cars performed better overall. We encourage creativity, innovation and competitive spirit by inspiring future teams to design their own custom cars.

2. Requirements

Below, we outline the essential requirements and tools needed.

2.1 Hardware

3D Printer

The mechanical components of the kit are designed to be simple enough to print as to not require expensive 3D printers. The printer we used is the **Snapmaker 2.0 A350T**, but any printer with a large enough bed should be good.

Motors, adapters and wheels

The setup consists of 1 servo motor for steering and 2 DC reductor motors with rear drive configuration featuring 3 mm diameter axles as shown in +. Figure 1.

On the axle we attach a hexagonal adapter onto which the wheels are mounted. Refer to Figure 2.

The hexagonal adapter measurements can be seen in Figure 3.

Note: Most motors without reductors do NOT have 3mm axles, they have smaller axles. If you want to use the same direct connection to the wheel with bigger motors, you will have to make/buy a different adapter.

Figure 1 Motors used in the kit.

Figure 2 Motor-Wheel connection.

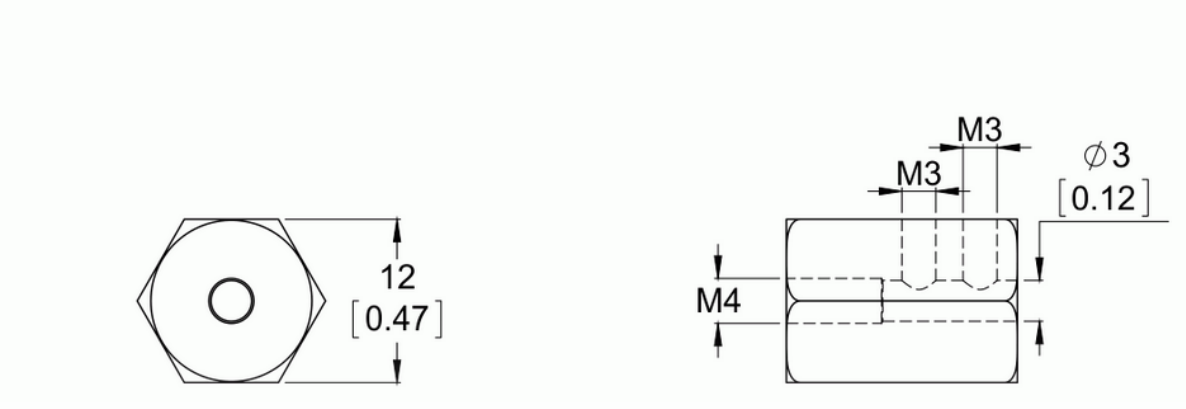


Figure 3 Hex adapter measurements

This straightforward motor-to-wheel connection eliminates the need for a complex transmission system, allowing for direct power transfer from the motors to the wheels.
However, it's important to note that we do not recommend reduction ratios larger than 30:1, as they will negatively impact the top speed. We encourage teams to experiment with different motors to find the optimal balance between speed and torque.
Additionally, while we designed the motor holders to be universal, our design may not fit great on all motor types. Teams are encouraged to modify the holders for their motor .

Motor Driver

1. H-Bridge (L298N): For smaller motors and/or reductors, this is a good, low-cost option. It is best suited for driving loads that do not exceed 2A.

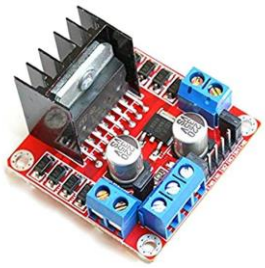


Figure 4 H-bridge

2. Electronic Speed Controller (ESC): For larger motors with higher current ratings and/or no reductors, an ESC will be better suited for handling the extra load. Some models are also capable of actively braking.

Steering Servo

For the best steering response time, a stronger servo capable of handling the extra load from tight turns is ideal. However, any servo should work, provided you can mount it.

Figure 5 Servo Motor

Battery

We recommend two-cell LiPo batteries with XT60 connectors, which meet the following requirements:
A minimum capacity of 1300mAh.
A nominal voltage of 7.4V.
30C discharge rate.

Figure 6 Battery

Camera

1. Pixy2 camera:
    - Specialized camera with an integrated processor designed for image processing tasks such as black line recognition.
    - It has been used in previous contests and is known for its good performance and reliability.
    - Plug-and-play solution with minimal setup but requires significant tinkering to get the best results.

Figure 7 Pixy2

2. TSL1401 Linear camera:
    - Cheaper and can perform acquisition much faster than a Pixy camera.
    - Less affected by external ambient light
    - No integrated line recognition software beyond the basic example provided in the sample app. The students will have to come up with their own solutions to interpret the raw analog data provided by the camera

Figure 8 Linear Camera

Development Board

Note that an NXP development board is needed for this competition.

The **S32K144-Q100** is a development board with RTD (Real Time Drivers) support, well-suited for this competition. This is the recommended board if you do not own one from previous competitions already.

Figure 9 S32K144-Q100

Note: the **RDDRONE-FMUK66**, also featured in past competitions, does not have any RTD support and as such we cannot provide you with any software support for it and will have to rely on the PX4 software stack if you want to use it.

2.2 Tools

Code IDE

S32 Design Studio is a complimentary Integrated Development Environment that enables editing, compiling and debugging of designs. This is the IDE used to configure, compile, run and debug any RTD software, and where you will write all the code for the competition. The installation process is described in sub-chapter 4.1 Design Studio IDE.

Link: S32 Design Studio IDE for Arm® based MCUs | NXP Semiconductors

G-Code generator

UltiMaker Cura is free, easy-to-use 3D printing software.

Link: https://ultimaker.com/software/ultimaker-cura/

CAD Software

Any Computer-aided design software that you prefer. Some examples that have educational licenses are Autodesk Inventor, Autodesk Fusion360.

If you do not have any CAD knowledge or want a programming-like 3D modelling software, try OpenSCAD. It is free and open source.

2.3 Real-Time Drivers

Real-Time Drivers (RTD) represent a cutting-edge and innovative set of drivers designed to support real-time software in both AUTOSAR® and non-AUTOSAR applications. These drivers are not just a tool, but a comprehensive framework and Application Programming Interface (API) that teams can leverage to control a 3D printed car.

The RTD are designed with a focus on real-time performance, ensuring that the software can respond instantly and predictably to changes in the environment or input signals. This is particularly crucial in automotive applications where timing and reliability can be critical for safety and functionality.

The RTD serves as the backbone for controlling a 3D printed car. They provide the necessary interfaces and functions that allow teams to interact with the car's systems.

3. Assembly

Below we outline the assembly of the kit. This will be divided in 2 parts, one for the mechanical structure and the other for the wiring.

Note: Prior to starting assembly, **make sure your servo is centered**.

3.1 Mechanical

Before starting the assembly make sure you have everything by laying down all the necessary components of the kit.

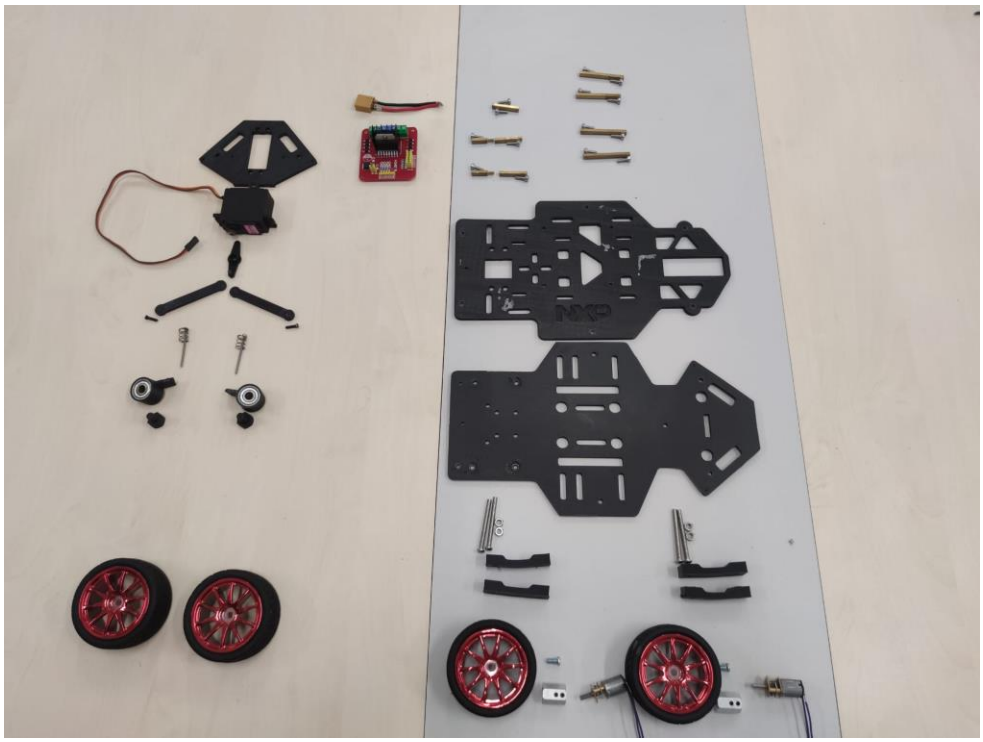


Figure 41 Top view of the parts

First, let's focus on the motors and their positioning. Add the M4 bolts at the backside of the plate.

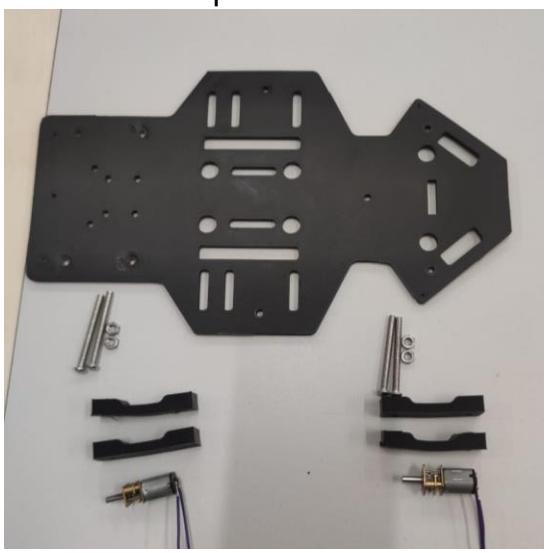


Figure 52 Motor holders

Figure 63 Adding bolts

Then we will hold the clamp with our fingers while we screw the bolts in..

Figure 74 Holding the clamp

Figure 85 Bolts fully screwed in

Next, we will position the motor and add the other clamp on top of it. Finally, we will hold everything together with 2 nuts.

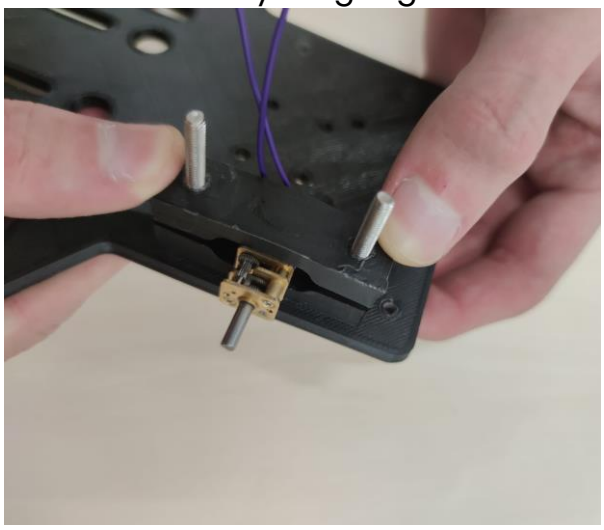


Figure 96 Motor positioning

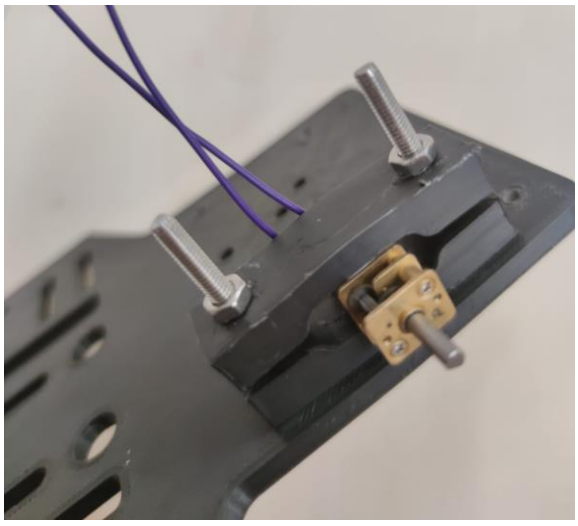


Figure 10 Motor attached

Follow the same instructions for the other side too

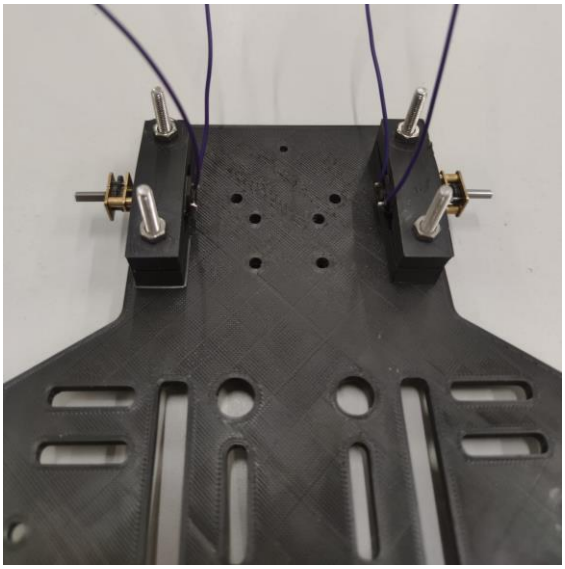


Figure 118 Second motor attached

Next, we will add the spacers. You can see in the photos below where every screw (the hexagonal ones) and spacer goes.

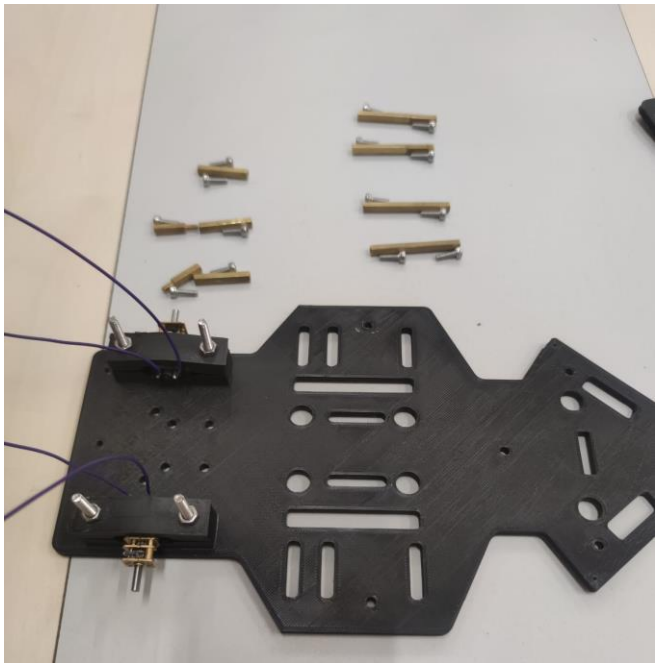


Figure 129 Spacers with bolts

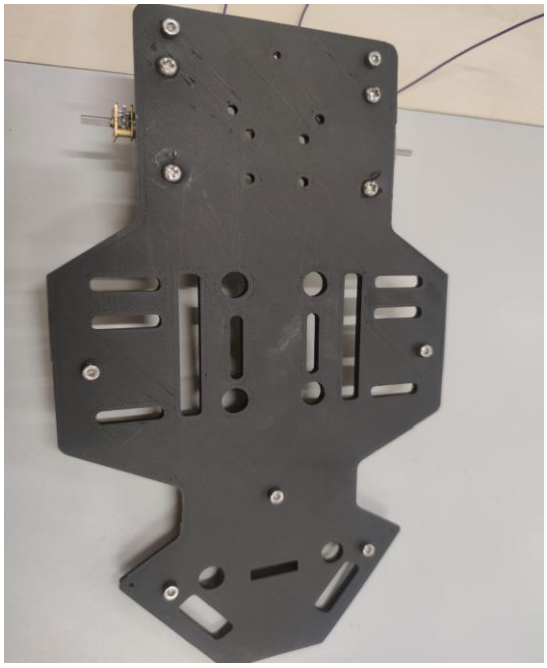


Figure 13 Bolts position

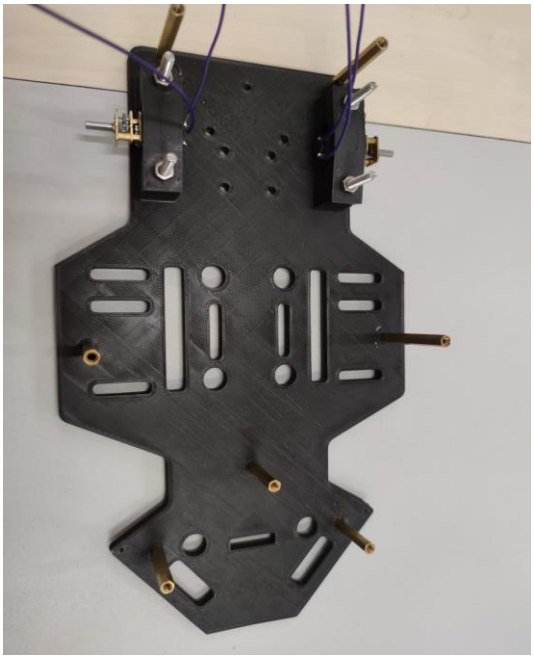


Figure 141 Spacers position

Now we can focus on the front of the car and its steering system.
Attention! The servo motor should be in its starting position before starting the assembly.
Screw the arm and the servo horn together. Then attach them to the servo motor.

Figure 152 The servo and arms

Figure 16 Servo horn and arms

Figure 174 Servo horn attached

Next let's insert the bearings in the bearing holders. Take your time here, try to lightly hammer them in.

Figure 185 Bearings and holders

Figure 19 Bearing inserted

Finally add the PLA axle that will hold the wheel.

Figure 207 PLA axle

Figure 21 axle inserted

We can continue and attach the servo to the wheel plate.

Figure 229 Servo and the wheel plate *Figure 23 servo attached*

Then add the whole assembly to the bottom plate with 1 bolt and the 2 15mm spacers.

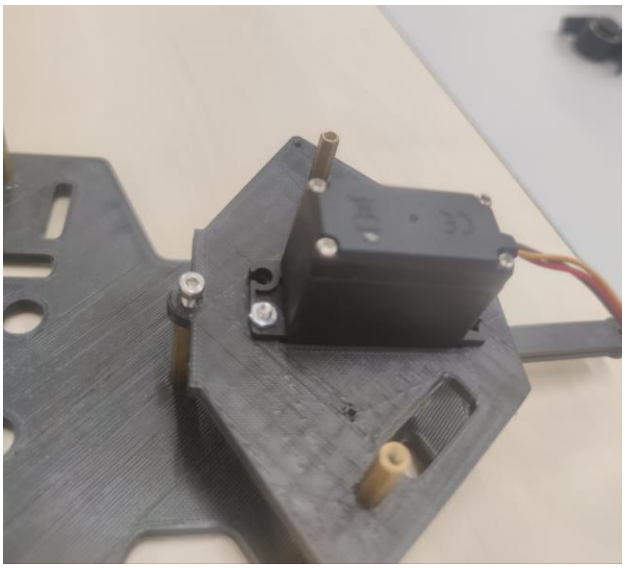


Figure 241 Assembly added to bottom plate

Now let's add the bearing holders.

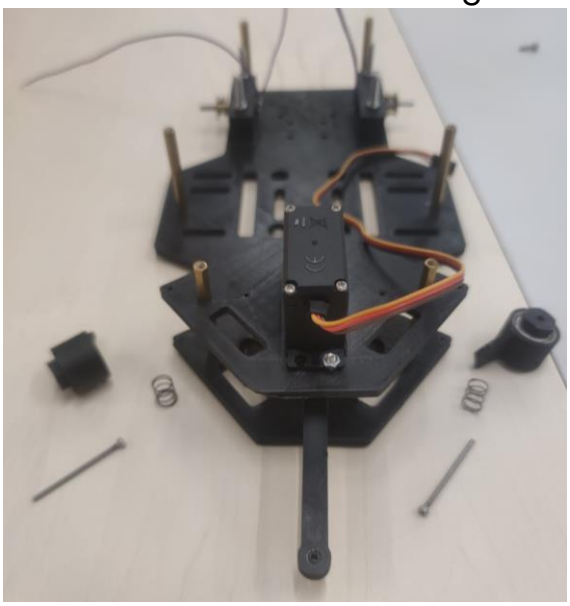


Figure 252 Preparing bearing holders

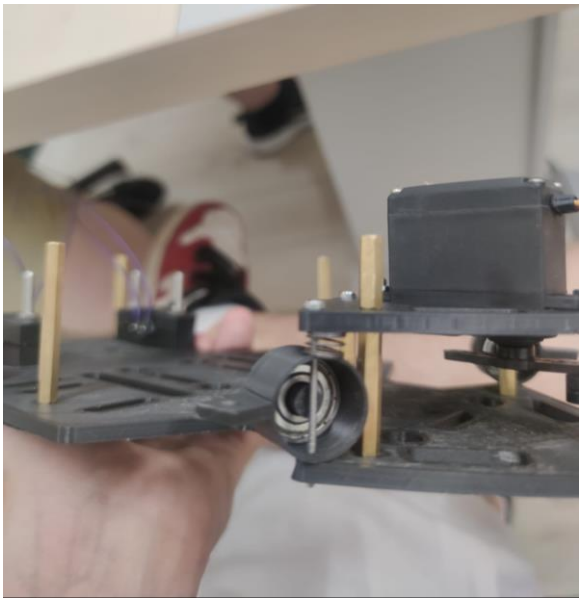


Figure 26 Bearing holder assembled

You need to position the spring and the bearing holder as shown in the picture, then insert the 36 mm screw through them. The spring is not on the shopping list, but you can obtain one from a pen and cut it to a suitable length. Do the same for the other side. Align the hole on the servo arm with the hole on the bearing holder then screw them together.

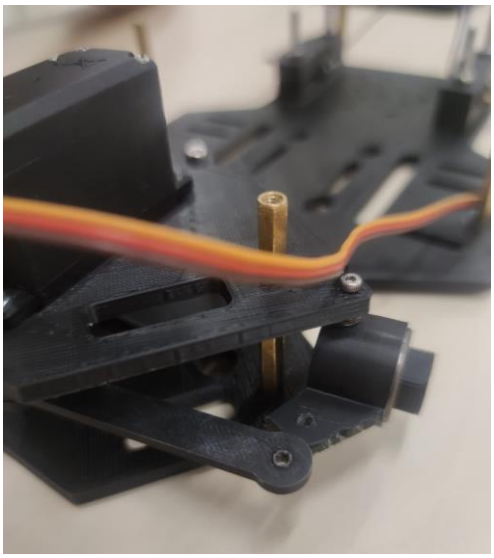


Figure 274 Align arm with bearing holder

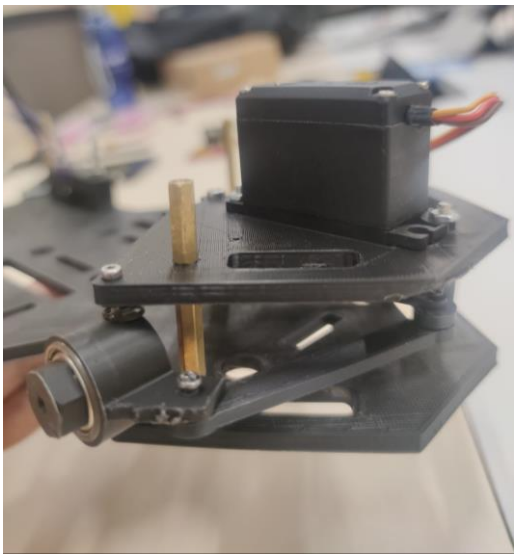


Figure 28 Screw them together

Now attach the hexagonal adapters on the motors and add the wheels.

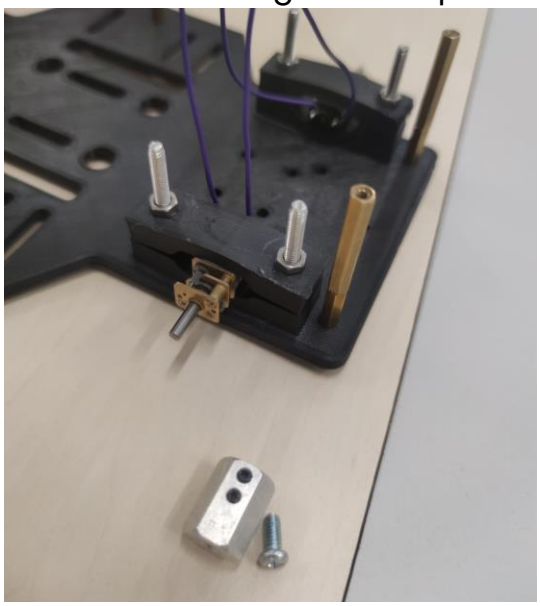


Figure 296 Preparing the adapter

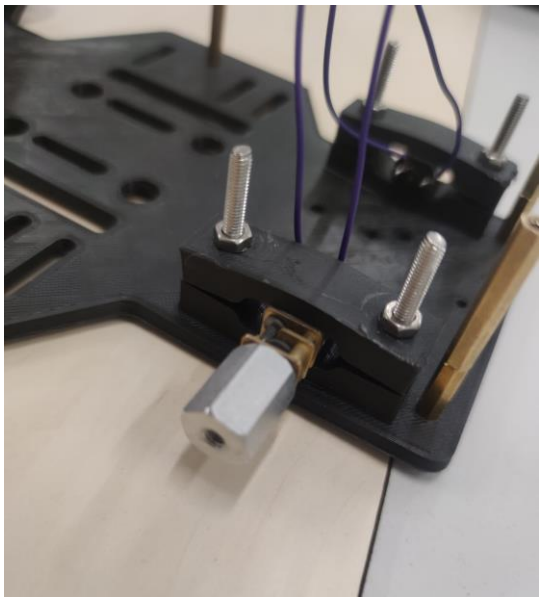


Figure 30 Attach the adapter

Figure 318 Preparing the wheels

Figure 329 Placing the wheels

Add the wheels in the front too.

Figure 330 Preparing the front wheels

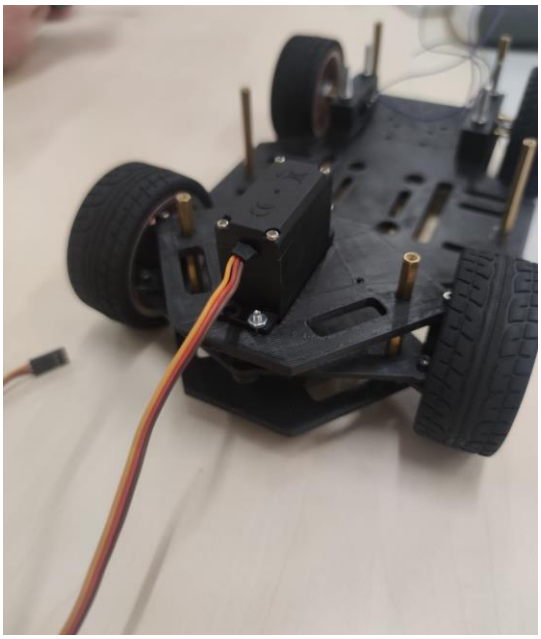


Figure 34 Placing the front wheels

Finally add the upper plate, screw it in and the mechanical structure is finished.

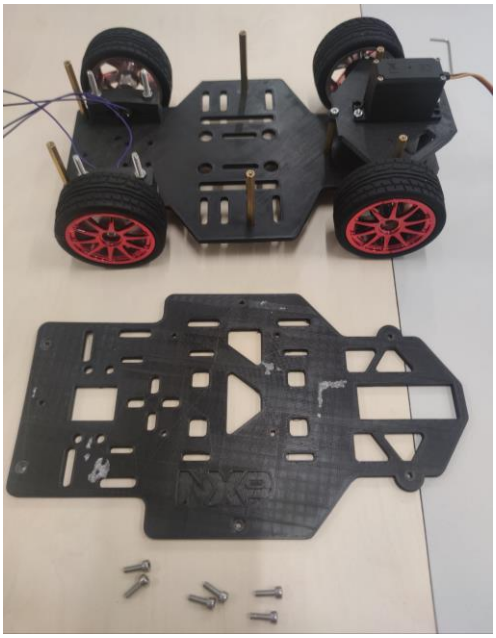


Figure 352 Preparing the upper plate

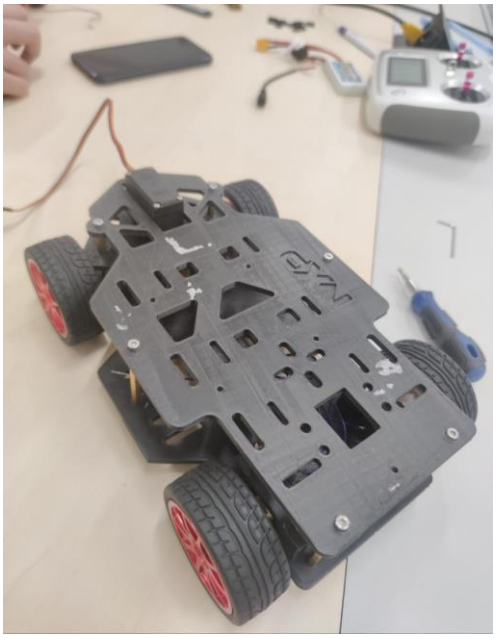


Figure 36 Finished car

3.2 Electric connections

Note: For some of these steps soldering tools are required.

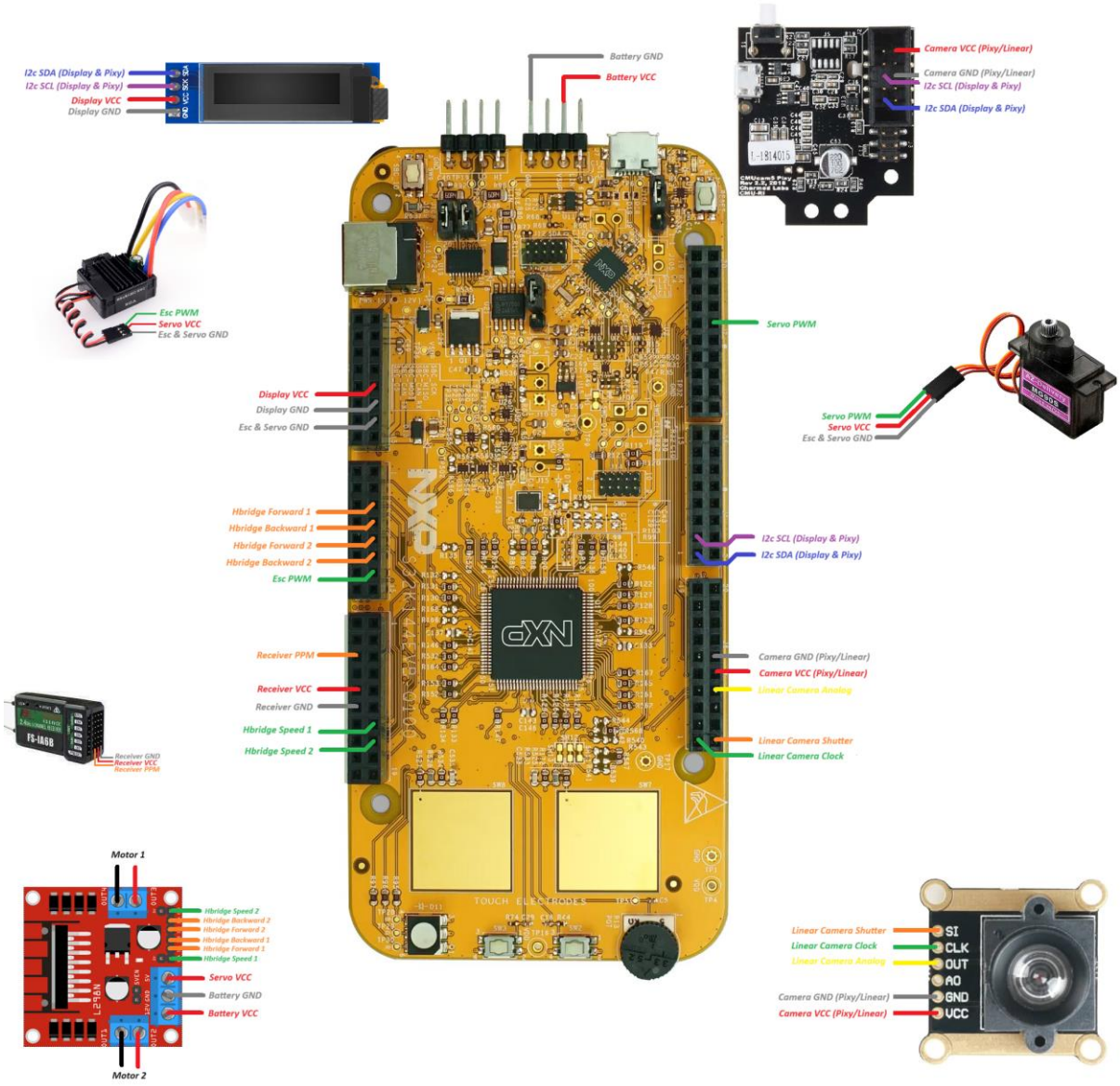


Figure 374 Board connections

Refer to Getting Started with the S32K144EVB | NXP Semiconductors
Here you will find a picture with all the board pins and pin names.

Component	PORTS/PINS connection	Notes
H-Bridge	PTC2 – Hbridge Speed 2 PTC0 - Hbridge Backward 2 PTA6 - Hbridge Forward 2 PTB1 - Hbridge Backward 1 PTB0 - Hbridge Forward 1 PTC1 - Hbridge Speed 1 From LIN communication Header: VBAT PIN: Battery VCC GND PIN: Battery GND From the H-Bridge&ESC: - Servo VCC(output)	Pins from the J4 and J5 headers.
Servo Motor	PTC13 - Servo pwm H-Bridge&ESC(out) - Servo vcc J3-13 - Servo & esc ground	Pins from the J2 and J3 headers.
ESC	PTE2 - Esc pwm H-Bridge&ESC(out) - Servo vcc J3-13 - Servo & esc ground	Pins from the J3 and J4 headers.
Linear Camera	PTD1 - LC Shutter PTD0 - LC Clock PTC16 - LC Analog J6-12 - Camera GND J6-10 - Camera VCC	LC means Linear Camera. Pins from the J6 header.
Pixy2 Camera	PTA3 – Pixy2&Display SCL PTA2 – Pixy2&Display SDA J6-12 - Camera GND J6-10 - Camera VCC	Pins from the J1 and J6 headers.
OLED Mini Display	PTA3 – Pixy2&Display SCL PTA2 – Pixy2&Display SDA J3-11 - Display GND J3-09 - Display VCC	For testing and debugging. Pins from the J3 and J6 headers.
Receiver	PTE14 - Receiver PPM J5-12 - Receiver GND J5-10 - Receiver VCC	For testing and debugging Pins from the J5 header.

As you can see, the OLED display and the Pixy2 camera share the same i2c pins. We will connect each SCL and SDA pin. The picture below shows one way of doing it. This must be done twice, once for the SCL pins and once for the SDA pins.

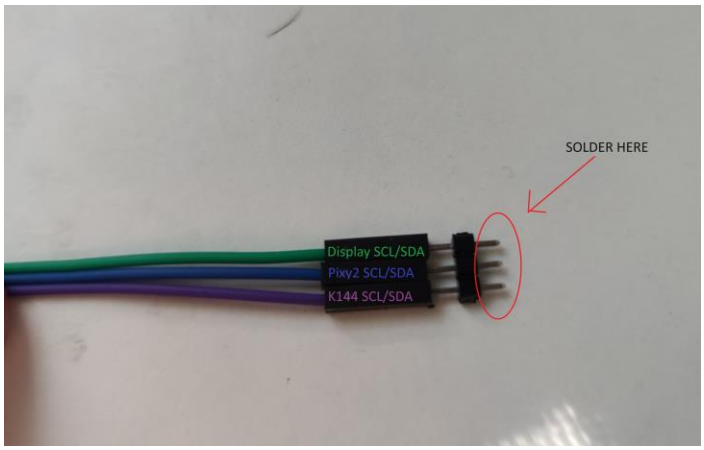


Figure 385 I2C connection. Purple wire goes to K144, green to display and blue to camera.

Power

For powering the board you will need the following xt60 connector. You can either buy one or make one yourself from two XT60 connectors. The red (vcc) and black(gnd) dupont wires will be connected to the LIN header pins, VBAT and GND as you can see in the picture below.

Figure 396 XT60 connector

Figure 407 powering the S32K144

The LiPo battery goes in the female port of the connector, and the ESC/Hbridge goes in the male port of the connector. Most ESC already

have a connector soldered, but for the Hbridge you will need two wires. Again, you will need an adapter which you can buy or make yourself.

Figure 48 Hbridge adapter

4. Software

4.1 Design Studio IDE

To install Design Studio IDE, an account on NXP site must be created. Please create an account at this link.

After the account has been created or if you already have an account on the NXP site, go to the Design Studio page found at this link.

The following steps show how to download and install the tool:

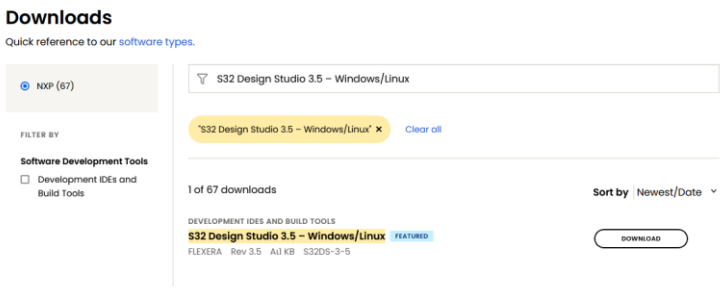
1. Click *S32DS for S32 Platform* link

S32 Design Studio IDE Editions

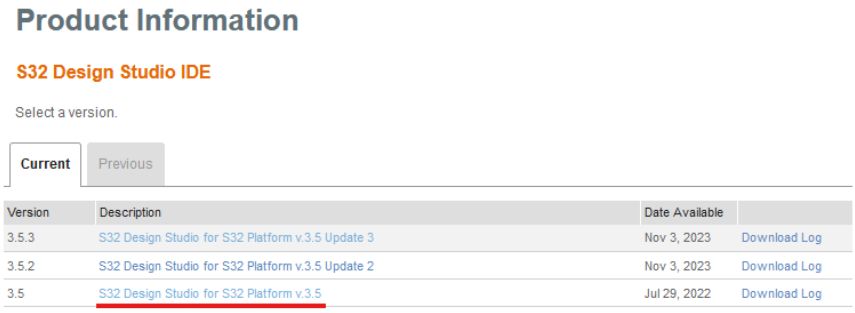
This map contains all supported content for all supported products.
Check release notes for details on which content is supported for a particular product.

	Supported Products	Integrated NXP Tools	Integrated NXP Software	Compilers	Debuggers	Dedicated Tools
S32DS for S32 Platform UPDATED	• S32G • S32K3 • S32K1 • S32S24 • S32Z2/E2 • S32V • S32R41 • S32R45	• S32 Configuration Tool ◦ Pins Wizard ◦ Clocks configuration ◦ Peripheral/ Drivers configuration ◦ DCD configuration ◦ IVT configuration ◦ QuadSPI configuration ◦ DDR configuration • S32 Flash Tool • FreeMASTER	• S32 RTD • S32 SDK • FreeRTOS • AMMCLib • Vision SDK • Linux® BSP	• NXP GCC 10.2* • NXP GCC 9.2 • NXP GCC 6.3.1* • Green Hills • IAR • WindRiver Diab	Built-in GDB interface: • S32 Debugger + S32 Debug Probe • P&E Multilink / Cyclone / Open SDA • SEGGER J-Link Lauterbach, iSystem and IAR supported	Vision: • NXP APU Compiler • ISP Assembler • ISP and APEX graph tools Radar: • SPT Visualization Tools (Timeline, Memory) • SPT assembler, explorer, graph tool

2. Click on the orange button *Downloads* on the new page
3. In the search bar type *S32 Design Studio 3.5 – Windows/Linux*



4. Click Download
5. Click on S32 Design Studio for S32 Platform v3.5 (last entry)



6. Download the *Windows Installer S32 Design Studio v3.5 Windows* installer and *S32 Design Studio v3.5 Installation Guide*
7. Follow the steps of the *S32 Design Studio v3.5 Installation Guide* for installing the tool

4.2 Software packet installation and prerequisites

The *official GitHub* repository where the NXP Cup RTD software is found is https://github.com/NXPHoverGames/NXP-CUP-Real-Time-Drivers-SW-bare-metal

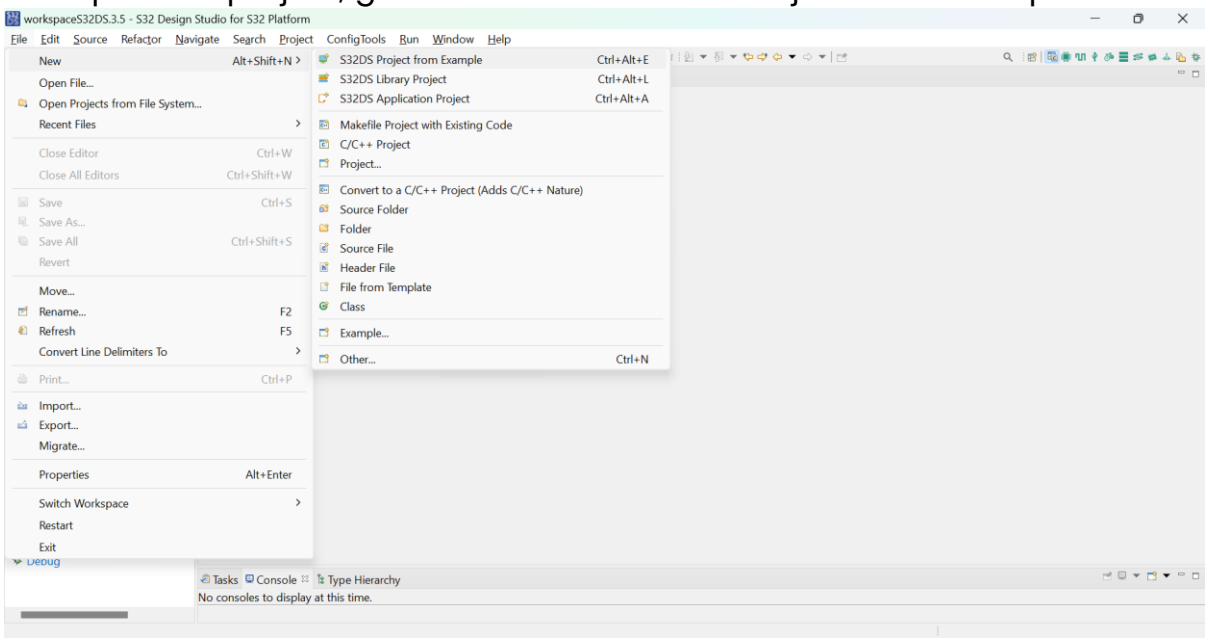
Download the .zip file on your computer and extract the files.

1. Start the Design Studio IDE. An S32DS Extensions and Updates Window will open.
2. Search for *S32K1xx development package* and *S32M2xx development package* and select them for installation.
3. Restart the tool after the installation is completed
4. Go to Help > Install New Software

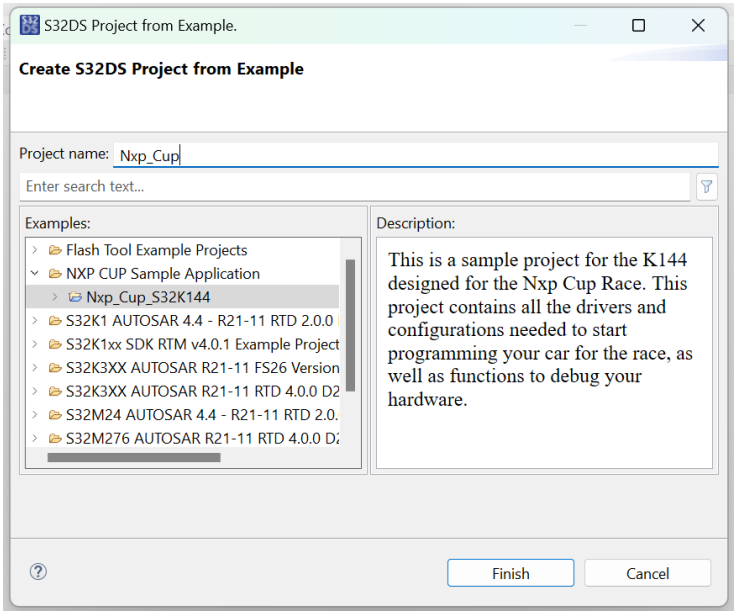
5. In the opened window click Add > Archive
6. Select the *SW32K1_S32M24x_RTD_4.4_R21-11_2.0.0_DS_updatesite_D2408.zip* from the downloaded GitHub repository.
7. Click Add and select all the files in the window. Proceed to install them.
8. After the installation is finished, restart the tool. The Real-Time drivers and the NXP Cup sample app are installed.

4.3 Importing the NXP Cup sample app

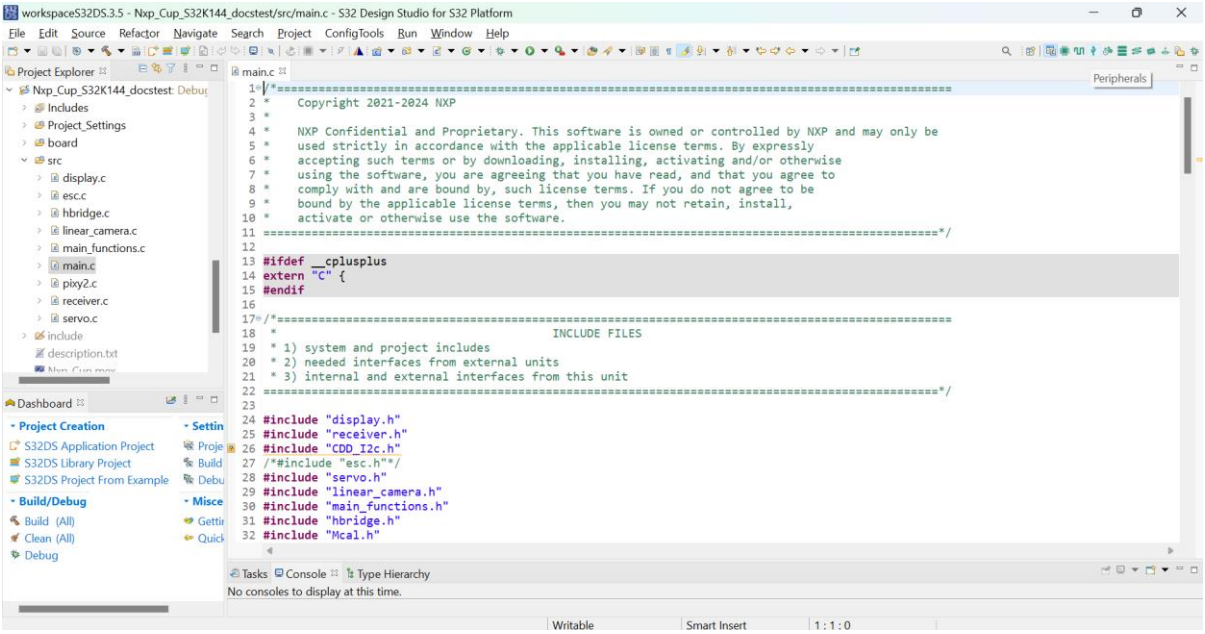
To import the project, go to File > New > S32DS Project from Example

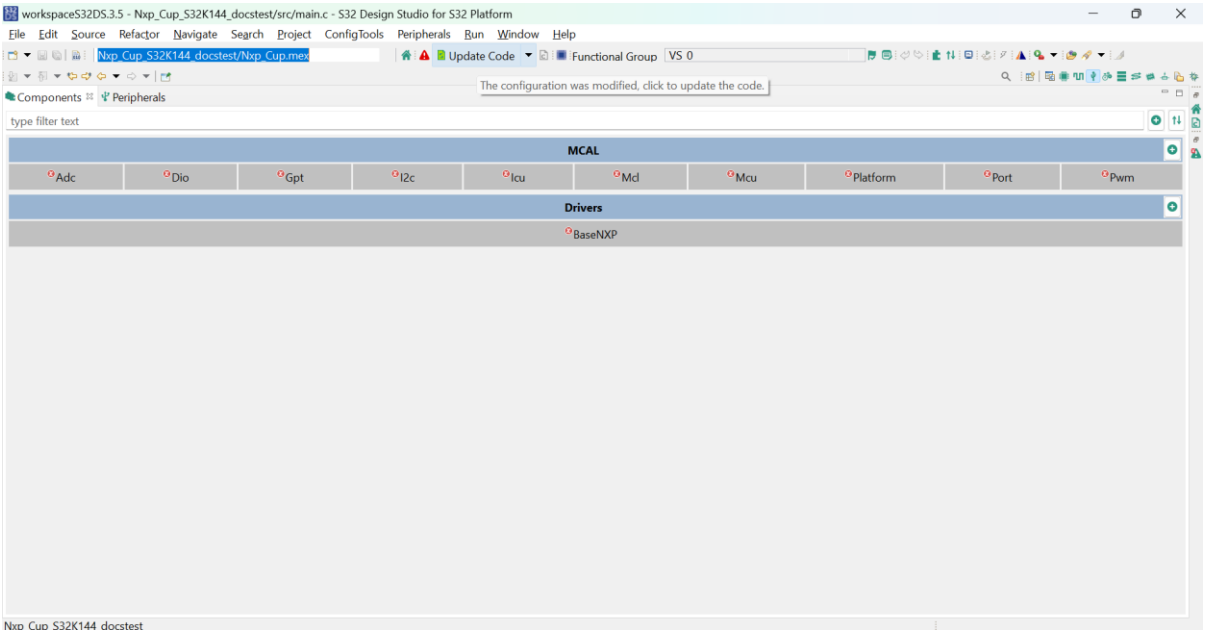


Inside the window that popped up, select NXP CUP Sample Application > Nxp_Cup_S32K144.

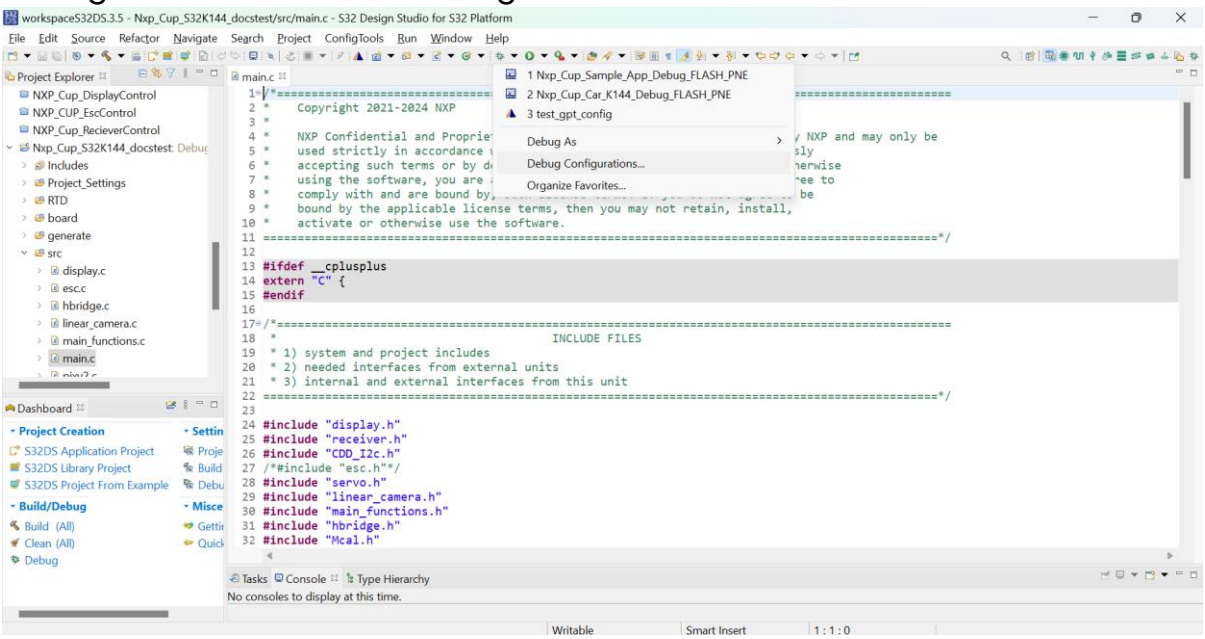


To generate the configuration of the drivers, go to Peripherals tab, select Update Code and press ok. **All errors in the Peripherals tool should go away after this step**. This step must be repeated when you change the configuration in the Peripherals, Clocks, Pins tools.

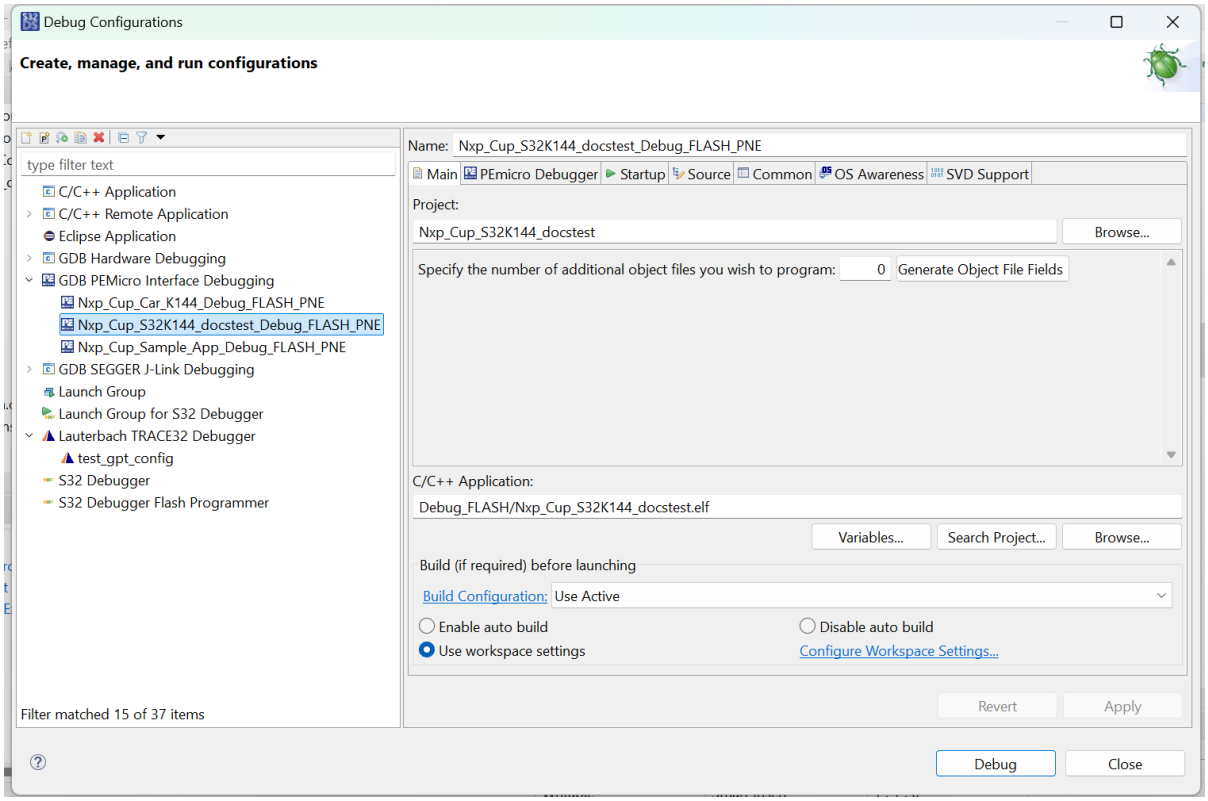




To run the code, go back to code view (C/C++ button) and press Debug Configurations under the Debug button menu.



Under GDB PEMicro Interface Debugging, select the one with your project's name. If there isn't one, create a new one. Press debug to upload the code.


Debug Configurations
Create, manage, and run configurations
type filter text
C/C++ Application
C/C++ Remote Application
Eclipse Application
GDB Hardware Debugging
GDB PEMicro Interface Debugging
Nxp_Cup_Car_K144_Debug_FLASH_PNE
Nxp_Cup_S32K144_docstest_Debug_FLASH_PNE
Nxp_Cup_Sample_App_Debug_FLASH_PNE
GDB SEGGER J-Link Debugging
Launch Group
Launch Group for S32 Debugger
Lauterbach TRACE32 Debugger
test_gpt_config
S32 Debugger
S32 Debugger Flash Programmer
Filter matched 15 of 37 items
Name: Nxp_Cup_S32K144_docstest_Debug_FLASH_PNE
Main
PEmicro Debugger
Startup
Source
Common
OS Awareness
SVD Support
Project:
Nxp_Cup_S32K144_docstest
Browse...
Specify the number of additional object files you wish to program:
0
Generate Object File Fields
C/C++ Application:
Debug_FLASH/Nxp_Cup_S32K144_docstest.elf
Variables...
Search Project...
Browse...
Build (if required) before launching
Build Configuration:
Use Active
Enable auto build
Disable auto build
Use workspace settings
Configure Workspace Settings...
Revert
Apply
Debug
Close

4.3 Drivers description and usage

The drivers provided in the example app will be explained in detail in this section, including how they work, how to use them and some improvements you can bring to them to get a competitive advantage during the race.

Note: Some drivers have test functions which you can use to ensure everything is connected and working properly. After you validated everything, remove them from your main code since these functions never return to main.

4.3.1 Servo

This driver is designed to control a servo motor via PWM (Pulse Width Modulation) signals. It allows for precise positioning of the servo arm by varying the duty cycle of the PWM signal.

Initialization:

```
void ServoInit(Pwm_ChannelType ServoPwmChannel, uint16 MaxDutyCycle, uint16 MinDutyCycle, uint16 MedDutyCycle )
```

Parameters:

- MaxDutyCycle: The maximum duty cycle, which corresponds to the servo's maximum rotation angle.
- MinDutyCycle: The minimum duty cycle, corresponding to the servo's minimum rotation angle.
- MedDutyCycle: The medium duty cycle, representing the neutral or center position of the servo.

The function sets the servo to its neutral position upon initialization.

Steering:

```
uint16 Steer(int Direction)
```

Parameters:

- Direction: Positive Direction values turn the servo right, and negative values turn it left.

Predefined Steering Actions:

```
void SteerLeft(void);
void SteerRight(void);
void SteerStraight(void);
```

- SteerLeft: Moves the servo to its maximum rotation angle (left).
- SteerRight: Moves the servo to its minimum rotation angle (right).
- SteerStraight: Aligns the servo to its neutral position (center).

4.3.2 ESC

This driver controls the speed and direction of a motor via an Electronic Speed Controller (ESC) using PWM (Pulse Width Modulation) signals. It includes a state machine to handle the motor's operation states, such as Forward, Reverse, Braking, and Neutral.

Initialization:

```
void EscInit(Pwm_ChannelType EscPwmChannel, uint16 MinDutyCycle, uint16 MedDutyCycle, uint16 MaxDutyCycle)
```

Parameters:

- EscPwmChannel: The PWM channel assigned to the ESC.
- MinDutyCycle: The minimum duty cycle, typically representing the lowest speed or a reverse command.
- MedDutyCycle: The medium duty cycle, usually corresponding to the neutral or idle state of the motor.
- MaxDutyCycle: The maximum duty cycle, indicating the highest speed or a forward command.

Upon initialization, the ESC is set to the neutral position, ready to receive speed commands.

Setting Speed:

```
void SetSpeed(int Speed)
```

Parameters:

- Speed: values between -100 and 100, Speed >= 0 means Forward, Speed < 0 means Reverse

Getting Speed:

```
int GetSpeed()
```
Get the speed value.

Braking:
```
void SetBrake(uint8 Brake)
```
Parameters:

- Brake: The car will brake if set to anything other than 0 (Ex. 1). Setting it to 0 disengages the brake and allows the car to speed up.

Getting Brake:
```
uint8 GetBrake()
```
Get the brake variable state.

Getting the state machine current state (for debug purposes):
```
enum EscStates GetEscState(void)
```

4.3.3 H Bridge

This driver controls the direction and speed of two motors using a L298N H-Bridge circuit, which allows for both forward and reverse movements through PWM (Pulse Width Modulation) signals. It also manages braking functionality.

Initialization:
```
void HbridgeInit(Pwm_ChannelType Motor1_Speed, Pwm_ChannelType Motor2_Speed, Dio_ChannelType Motor1_Forward,
       Dio_ChannelType Motor1_Backward, Dio_ChannelType Motor2_Forward, Dio_ChannelType Motor2_Backward){
```
Parameters:

- Motor1_Speed & Motor2_Speed: PWM channels for controlling the speed of Motor 1 and Motor 2.
- Motor1_Forward & Motor1_Backward: Digital I/O channels for setting the direction of Motor 1.
- Motor2_Forward & Motor2_Backward: Digital I/O channels for setting the direction of Motor 2.

Speed Control:
```
void HbridgeSetSpeed(int Speed)
```

Parameters:

- Speed: An integer value between -100 and 100, where positive values indicate forward motion and negative values indicate reverse motion.

The function calculates the duty cycle based on the speed and sets the direction of the motors accordingly. If the handbrake is not engaged, it will apply the calculated duty cycle to the motors' speed channels.

Braking:

```
void HbridgeSetBrake(uint8 Brake)
```

Parameters:

- Brake: The car will brake if the parameter is not 0.

Getting Speed:

```
int HbridgeGetSpeed()
```

Get the speed value.

Getting Brake:

```
uint8 HbridgeGetBrake()
```

Get the brake variable state.

4.3.4 Display

This driver is for the display that comes with the MR-CANHUBK344 board. It is a 128x32 OLED display with a SSD1306 chip with I2c. **The chip is also found in 128x64 I2c displays, and the driver is easy to port for them**.

The driver divides the pixels into **4 lines of 16 characters each** using an 8x8 character fontmap. You can print text, positive and negative integers and waveforms with this driver. The driver keeps a buffer of all the display's pixels. The current implementation requires manual display refresh.

Initializing the driver:

```
void DisplayInit(uint8 FontmapRotations);
```

Parameters:

- FontmapRotations: How many times should the fontmap be rotated. This parameter should be kept as 1 in most circumstances.

Printing text:

```
void DisplayText(uint16 DisplayLine, const char Text[16], uint16 TextLength, uint16 TextOffset);
```

Parameters:

- DisplayLine: The line on which to print the text.
- Text[16]: The actual text to print.
- TextLength: The length of the text to print.
- TextOffset: How many positions from the start of the line (left) to skip before printing the text.

Printing positive/negative integers:

```
void DisplayValue(uint16 DisplayLine, int Value, uint16 TextLength, uint16 TextOffset);
```

Parameters:

- DisplayLine: The line on which to print the number.
- Value: The actual number to print.
- TextLength: How many digits to print. If this parameter is larger than the number of digits the number has, it will be completed with blank spaces. If it is smaller, the number will be cropped to fit.
- TextOffset: How many positions from the start of the line (left) to skip before printing the number.

Printing waveforms:

```
void DisplayGraph(uint8 DisplayLine, uint8 Values[128], uint16 ValuesCount, uint8 LinesSpan);
```

Parameters:

- DisplayLine: The line on which to start printing the waveform.
- Values[128]: The actual values to print. The values must be in the 0-100 range.
- ValuesCount: How many values to print. This also determines how many pixels the waveform will occupy horizontally on the display.
- LinesSpan: How many **lines** should the waveform occupy **vertically**, starting from the DisplayLine parameter and going down. The image will be scaled automatically to fit.

Clearing display: This function clears the display's buffer. It still needs a refresh to see the changes.

```
void DisplayClear(void);
```

Refreshing display: This function sends to the display every change made to the buffer.

```
void DisplayRefresh(void);
```

4.3.5 PPM Receiver

This driver is meant to be used with a FS-IA6B Receiver set in PPM mode, or any other 8-channel receiver with PPM support. The driver decodes 8 channels and makes their values available for various uses. Each of the eight channels are mapped to different switches and actuators on the RC remote. The driver automatically decodes the channels after initialization.

Initialization:

```
void ReceiverInit(Icu_ChannelType ReceiverIcuChannel, Gpt_ChannelType ReceiverGptChannel,
        uint16 ChannelMinTicks, uint16 ChannelMedTicks, uint16 ChannelMaxTicks, uint16 OutsideChannelTicks);
```

Parameters:

- ReceiverIcuChannel: The Icu channel configured for the receiver in Peripherals tool.
- ReceiverGptChannel: The Gpt channel configured for the receiver in Peripherals tool.
- ChannelMinTicks, ChannelMedTicks, ChannelMaxTicks, OutsideChannelTicks: The number of ticks for shortest, median and longest PPM signal length, as well as minimum signal length for the idle signal between channels. These values should be left as is unless the driver clocks, prescalers etc are changed.

Getting receiver state: Useful for debugging purposes

```
enum ReceiverStates GetReceiverState(void);
```

Parameters: None

Getting receiver channel values: The values are typically between -100 and 100, with two exceptions: when the receiver is not connected/working, it

returns -200; small timing issues due to software decoding can add or subtract small amounts, going past the -100 to 100 range.

```
int GetReceiverChannel(uint8 Channel);
```

Parameters:

- Channel: The channel for which to return the value.

4.3.6 Linear Camera

The Linear camera driver is suitable for 128x1 cameras. It automatically sends the clock and shutter signals to the camera and stores the analog values in an array.

Note: The driver does not provide a function to get the values. Instead, you should declare the buffer as an extern variable to get the values directly from the driver.

Initialization:

```
void LinearCameraInit(Pwm_ChannelType ClkPwmChannel, Pwm_ChannelType ShutterPwmChannel, Adc_GroupType InputAdcGroup
```

Parameters:

- ClkPwmChannel: The Pwm channel configured for the camera in the Peripherals tool.
- ShutterPwmChannel: The Pwm channel configured for the camera in the Peripherals tool.
- InputAdcGroup: The Adc Group configured for the camera in the Peripherals tool.

4.3.6 Pixy2 Camera

The Pixy2 camera driver communicates via I2c to get all the detected vectors and their indexes.

Initialization:

```
void Pixy2Init(I2c_AddressType PixyI2cAddress, uint8 PixyI2cChannel);
```

Parameters:

- PixyI2cAddress: The I2c address configured on the Pixy2 via the Pixymon app.
- uint8 PixyI2cChannel: The I2c channel configured for the camera in the Peripherals tool.

Setting the LED on the Pixy: Useful for debugging purposes

```
void PixySetLed(uint8 Red, uint8 Green, uint8 Blue);
```

Parameters:

- uint8 Red, Green, Blue: The red, green and blue values of the desired LED color.

Getting the detected vectors: The returned structure contains the number of vectors, and the actual vectors along with their indexes.

```
DetectedVectors PixyGetVectors(void);
```

The returned structure DetectedVectors:

```
typedef struct {
    uint8 NumberOfVectors;
    Vector Vectors[100];
} DetectedVectors;
```

The Vector Structure:

```
typedef struct{
    uint16 x0, y0, x1, y1;
    uint8 VectorIndex;
}Vector;
```

Note: The GetPixy2State function will be used in a future version of the driver and should not be called.

```
Pixy2State GetPixy2State(void);
```

5. References

Figure 1 Motor micro DC, Robofun, GA12-N20, 6V, 400 RPM, cu angrenaje metalice - eMAG.ro
Servomotor MG996R Tower Pro - Moviltronics
Figure 2 Adaptor pentru ax 3mm lungime 12mm (robofun.ro)
Figure 3 Adaptor pentru ax 3mm lungime 12mm (robofun.ro)
Figure 4 L298N Motor Driver Module Pinout, Datasheet, Features & Specs (components101.com)
Figure 5 Servomotor MG996R Tower Pro - Moviltronics
Figure 6 Acumulator Gens Ace G-Tech Soaring 1300mAh 7.4V 30C 2S1P XT60 - eMAG.ro
Figure 7 Pixy2 – PixyCam
Figure 8 TSL1401 Linescan Camera Module - Micro Robotics
Figure 9 S32K144-Q100 Evaluation Board for Automotive General Purpose | NXP Semiconductors
Figure 10 MR-CANHUBK344 Product Information|NXP